7. 통계 테이블 설계

#0.강의/2.데이터베이스로드맵/4.설계2

통계 데이터와 성능 문제

우리가 만드는 쇼핑몰 스타트업이 대박이 났다. 주문이 하루에 수십만 건, 수백만 건씩 쌓이고 있다. 매출은 늘어나고 데이터베이스에 데이터가 가득 차고 있는 행복한 상황이다.

그런데 어느 날 기획팀과 대표님이 찾아와서 이런 요구를 한다.
"매일매일 일별 매출 현황을 보고 싶어요. 그리고 월별 매출 추이도 그래프로 보고 싶습니다."

개발자인 우리는 아주 자연스럽게 "네, 금방 만들어 드리겠습니다."라고 대답한다. 그리고 바로 SQL을 작성하기 시작한다. 가장 먼저 떠오르는 방법은 무엇인가? 바로 원본 주문 테이블(`orders`)을 날짜별로 `GROUP BY` 해서 집계하는 것이다.

왜 이 방법이 문제가 되는지, 그리고 실무에서 어떤 일이 벌어지는지 직접 확인해보자.

원본 데이터 준비

먼저 상황을 재현하기 위해 주문 테이블을 만들고 데이터를 입력해보자. 주문 테이블에는 기본적인 주문 정보들이 포함된다.

```
DROP TABLE IF EXISTS orders;

CREATE TABLE orders (
```

```
    order_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    customer_id BIGINT NOT NULL,
    total_amount INT NOT NULL,
    order_status VARCHAR(20) NOT NULL,
    order_date DATETIME NOT NULL
);

-- 데이터 입력 (예시를 위해 소량만 입력하지만, 실제로는 수백만 건이라고 가정한다)
INSERT INTO orders (customer_id, total_amount, order_status, order_date)
VALUES
(1, 10000, 'COMPLETED', '2026-01-01 10:00:00'),
(2, 25000, 'COMPLETED', '2026-01-01 14:30:00'),
(3, 15000, 'COMPLETED', '2026-01-01 18:20:00'),
(4, 50000, 'COMPLETED', '2026-01-02 09:15:00'),
(5, 30000, 'COMPLETED', '2026-01-02 11:00:00'),
(6, 12000, 'CANCELLED', '2026-01-02 15:45:00'),
(7, 45000, 'COMPLETED', '2026-01-03 10:00:00');
```

```
SELECT * FROM orders;
```

[실행 결과]

order_id	customer_id	total_amount	order_status	order_date
1	1	10000	COMPLETED	2026-01-01 10:00:00
2	2	25000	COMPLETED	2026-01-01 14:30:00
3	3	15000	COMPLETED	2026-01-01 18:20:00
4	4	50000	COMPLETED	2026-01-02 09:15:00
5	5	30000	COMPLETED	2026-01-02 11:00:00
6	6	12000	CANCELLED	2026-01-02 15:45:00
7	7	45000	COMPLETED	2026-01-03 10:00:00

원본 테이블 직접 집계의 유혹

요구사항은 일별 매출 통계다.

`DATE()` 함수로 날짜만 사용하고, `GROUP BY`를 사용한다. 취소된 주문은 제외해야 하니 `WHERE` 조건도 들어간다.

```
SELECT
    DATE(order_date) as stat_date,
    COUNT(*) as order_count,
    SUM(total_amount) as total_sales
FROM orders
WHERE order_status = 'COMPLETED'
GROUP BY DATE(order_date)
ORDER BY stat_date;
```

[실행 결과]

stat_date	order_count	total_sales
2026-01-01	3	50000
2026-01-02	2	80000
2026-01-03	1	45000

결과가 아주 잘 나온다. 개발 서버에서 테스트할 때는 데이터가 몇 개 없으니 속도도 0.001초면 나온다. "완벽해!"라고 생각하고 배포한다. 이것이 주니어 개발자가 가장 많이 하는 실수다.

왜 원본 직접 집계가 위험한가?

서비스 초기에는 아무 문제가 없다. 하지만 쇼핑몰이 성장해서 `orders` 테이블에 데이터가 1,000만 건, 1억 건이 쌓이면 어떻게 될까?

1. **성능 저하**: 위 쿼리는 기본적으로 인덱스를 타기 어렵거나, 타더라도 범위가 매우 넓다.
2. **데이터베이스 부하**: 통계 쿼리는 보통 처리해야 하는 행이 매우 많으므로 CPU와 메모리를 엄청나게 많이 사용한다. 이 무거운 쿼리가 도는 동안 데이터베이스의 자원을 아주 많이 사용한다.
3. **서비스 장애 전파**: 이게 제일 무서운 점이다. 관리자가 "매출 좀 볼까?" 하고 조회 버튼을 누르는 순간, DB가 통계 쿼리를 처리하느라 바빠져서 정작 고객이 상품을 구매하고 결제하는 `INSERT`, `UPDATE` 쿼리가 점점 느려지거나 최악의 경우 타임아웃이 발생한다.

결국 실무에서는 "통계 조회는 사용자가 없는 새벽에만 하세요"라는 슬픈 공지가 내려오거나, "조회 버튼 누르면 DB 뻗으니 누르지 마세요"라는 경고문이 붙게 된다.

우리는 이 문제를 해결해야 한다. 대용량 트래픽을 처리하는 아키텍처에서 통계는 원본 테이블과 분리되어야 한다.

통계 테이블 설계

앞서 본 문제를 해결하는 가장 확실한 방법은 **통계를 위한 별도의 테이블**을 만드는 것이다. 이를 보통 '요약 테이블(Summary Table)' 또는 '통계 테이블'이라고 부른다.

핵심 아이디어는 간단하다. "미리 다 계산해서 저장해두자"는 것이다.

통계 테이블 생성

일별 매출 통계를 저장할 전용 테이블을 설계해보자. 원본 데이터에서 필요한 집계 정보만 뽑아서 저장한다.

```
DROP TABLE IF EXISTS daily_sales_stats;

CREATE TABLE daily_sales_stats (
    stat_date DATE NOT NULL,
    total_order_count INT NOT NULL DEFAULT 0,
    total_sales_amount BIGINT NOT NULL DEFAULT 0,
    PRIMARY KEY (stat_date)
);
```

이 테이블의 특징을 보자.

- `stat_date`: 통계 기준 날짜다. 하루에 딱 한 줄만 생긴다.
- `BIGINT`: 매출액은 커질 수 있으니 넉넉하게 잡는다.
- **데이터 양**: 하루에 딱 한 행만 만들어지기 때문에 1년이 지나도 365행, 10년이 지나도 3,650행이다. 원본이 1억 건이라도 이 테이블은 수천 건에 불과하다.

통계 데이터 생성 (배치 처리)

이제 원본 테이블(`orders`)에서 데이터를 읽어와서 통계 테이블(`daily_sales_stats`)에 넣어주어야 한다. 보통 이런 작업은 하루가 지난 다음 날 새벽(사용자가 거의 없는 시간)에 '배치(Batch)' 프로그램으로 실행한다.

예를 들어, 2026년 1월 1일이 지나고 1월 2일 새벽에 1월 1일치 데이터를 집계해서 넣는 것이다.

```
-- 1월 1일 00시부터 1월 2일 00시 직전까지의 데이터 (1월 1일 전체)
INSERT INTO daily_sales_stats (stat_date, total_order_count,
total_sales_amount)
SELECT
    DATE(order_date),
    COUNT(*),
    SUM(total_amount)
FROM orders
WHERE order_date >= '2026-01-01 00:00:00'
  AND order_date < '2026-01-02 00:00:00'
  AND order_status = 'COMPLETED'
GROUP BY DATE(order_date);
```

- 하루가 지나고 2026년 1월 2일이 되었다고 가정하자.
- 이때 하루 전인 2026년 1월 1일 데이터의 통계 데이터를 한 행으로 생성하는 것이다.

데이터가 잘 들어갔는지 확인해보자.

```
SELECT * FROM daily_sales_stats;
```

[실행 결과]

stat_date	total_order_count	total_sales_amount
2026-01-01	3	50000

> **배치 실행 시간**
> 다음날인 0시 0분 0초 이후에 배치를 실행하면 데이터에는 문제가 없다.
> 하지만 그 시간에도 아직 사용자가 많아서 서비스에 영향을 줄 수 있다면, 시스템을 모니터링 해보고 사용자가 거의 없는 시간을 찾아서 선택하면 된다.

> 글로벌 서비스가 아니라면 대부분 새벽 3시 ~ 5시 사이가 될 것이다.

통계 테이블 조회

이제 관리자 페이지의 매출 조회 기능은 원본 `orders` 테이블이 아니라, 이 `daily_sales_stats` 테이블을 바라보게 바꾼다.

```
SELECT * FROM daily_sales_stats WHERE stat_date = '2026-01-01';
```

[실행 결과]

stat_date	total_order_count	total_sales_amount
2026-01-01	3	50000

장점과 단점

장점

- **조회 속도**: 1억 건을 뒤지는 게 아니라 단 1건(또는 범위 조회 시 수십 건)만 읽으면 된다. 인덱스(`stat_date`)를 통해 0.001초 만에 결과가 나온다.
- **부하 분리**: 무거운 집계 쿼리는 새벽에 한 번만 실행된다. 업무 시간에는 가벼운 통계 조회 쿼리만 실행되므로 서비스 DB에 전혀 영향을 주지 않는다.

단점

- **실시간성 부족**: 어제까지의 데이터만 볼 수 있다. "오늘 지금 매출 얼마예요?"라는 질문에는 답할 수 없다. (이 문제는 뒤에서 해결할 것이다.)
- **관리 포인트 증가**: 배치 프로그램을 만들고 관리해야 한다. 배치가 실패하면 통계가 갱신되지 않는다.

주간, 월간 통계의 효율적인 처리

일별 통계(`daily`)는 만들었다. 그런데 기획자가 와서 또 묻는다.
"주간 통계랑 월간 통계 테이블도 따로 만들어야 하나요? `weekly_sales_stats`, `monthly_sales_stats` 처럼 요?"

이 질문에 대해 고민해보자. 과연 필요할까?

왜 일별 데이터면 충분한가?

결론부터 말하면 **대부분의 경우 일별 통계 테이블 하나면 충분하다.** 굳이 월간, 연간 테이블을 따로 만들 필요가 없다.

왜 그럴까? 데이터의 양을 생각해보자.

- 일별 통계 테이블(`daily_sales_stats`)에 10년치 데이터가 쌓여 있다고 가정해보자.
- 행의 개수: 3,650개.
- 데이터베이스 입장에서 3,650개 행을 `GROUP BY` 해서 월별로 합치는 것은 일도 아니다. 비용이 거의 '0'에 수렴한다.

그러므로 중복해서 월간 테이블을 만들고 관리 포인트를 늘리는 것보다, 일별 테이블을 재집계하는 것이 훨씬 효율적이다.

일별 통계를 활용한 월간 통계 조회

`daily_sales_stats` 테이블을 이용해서 2025년 11월 ~ 2026년 1월의 월간 매출을 조회해보자.

먼저 테스트를 위해 2025년 11월, 12월 데이터도 통계가 만들어졌다고 가정하고, 통계 테이블에 넣어두겠다. (실제로는 배치로 들어갔을 것이다)

```
-- 2025년 11월, 12월 데이터 강제 입력
INSERT INTO daily_sales_stats VALUES ('2025-11-01', 1, 10000);
INSERT INTO daily_sales_stats VALUES ('2025-11-02', 2, 15000);
INSERT INTO daily_sales_stats VALUES ('2025-12-01', 2, 20000);
INSERT INTO daily_sales_stats VALUES ('2025-12-02', 3, 21000);
```

이제 이 `daily_sales_stats` 테이블만 가지고 월별 통계를 뽑아보자.

```sql
SELECT
    DATE_FORMAT(stat_date, '%Y-%m') as stat_month,
    SUM(total_order_count) as monthly_order_count,
    SUM(total_sales_amount) as monthly_sales_amount
FROM daily_sales_stats
WHERE stat_date >= '2025-11-01' AND stat_date < '2026-02-01'
GROUP BY DATE_FORMAT(stat_date, '%Y-%m');
```

[실행 결과]

stat_month	monthly_order_count	monthly_sales_amount
2025-11	3	25000
2025-12	5	41000
2026-01	3	50000

일별 통계를 활용한 연간 통계 조회

월간 통계를 구하는 것과 원리는 완전히 같다. `GROUP BY`의 기준을 월(`%Y-%m`)에서 연(`%Y`)으로 바꾸기만 하면 된다.

2025년부터 2026년까지의 연간 매출 데이터를 조회해보자.

```sql
SELECT
    DATE_FORMAT(stat_date, '%Y') AS stat_year,
    SUM(total_order_count) AS yearly_order_count,
    SUM(total_sales_amount) AS yearly_sales_amount
FROM daily_sales_stats
WHERE stat_date >= '2025-01-01' AND stat_date < '2027-01-01'
GROUP BY DATE_FORMAT(stat_date, '%Y');
```

[실행 결과]

stat_year	yearly_order_count	yearly_sales_amount
2025	8	66000

2026	3	50000

2025년 데이터는 앞서 넣었던 11월, 12월 데이터가 합산되었고, 2026년 데이터는 1월 데이터가 합산된 것을 확인할 수 있다.

결국 **'일별 통계 테이블(`daily_sales_stats`)'** 하나만 잘 설계해두면, 주간, 월간, 분기, 연간 등 시간과 관련된 모든 통계를 자유자재로, 그리고 아주 빠르게 뽑아낼 수 있다. 이것이 통계 DB 설계의 핵심이다.

실무에서는 이러한 데이터를 활용해 대시보드에서 연도별 성장률을 그래프로 그리거나, 작년 대비 매출 비중을 분석하는 용도로 사용한다. 원본 데이터를 직접 건드리지 않기 때문에 대시보드를 새로고침할 때마다 서비스 DB가 멈출까 봐 걱정할 필요가 전혀 없다.

정리

- **원본(1억 건) → 일별 통계(수천 건)**: 데이터가 획기적으로 줄어든다.
- **일별 통계 → 월별/연간 통계**: 데이터 양이 이미 충분히 작다. 쿼리로 해결 가능하다. 굳이 별도 테이블을 만들지 마라.
- 단, 서비스 규모가 글로벌 급으로 너무 커서 다양한 종류의 일별 통계가 있고 또 수억 건이 된다면 그때는 월별 테이블이 필요하겠지만, 일반적인 서비스에서는 일별 통계면 충분하다.

실시간 통계와 하이브리드 설계

앞선 설계로 어제까지의 통계는 아주 빠르고 효율적으로 제공할 수 있게 되었다. 그런데 대표님이 다시 찾아왔다.

"어제 매출은 잘 보이는데, **오늘 지금 이 순간** 매출이 궁금해요. 실시간 매출 대시보드가 필요합니다."

난감하다.

1. `daily_sales_stats`에는 오늘 데이터가 없다. (내일 새벽에 들어오니까)
2. 그렇다고 원본 `orders`를 `GROUP BY` 하자니 아까 말한 성능 문제가 다시 발생한다.

이럴 때 사용하는 것이 바로 **통계 테이블과 원본 테이블을 결합한 하이브리드 방식**이다.

하이브리드 조회 전략

전략은 다음과 같다.

1. **과거 데이터**: 이미 집계된 `daily_sales_stats`에서 가져온다. (매우 빠름)
2. **오늘 데이터**: 원본 `orders` 테이블에서 오늘 데이터만 가져온다. (데이터 양이 적음)
3. **결합**: 이 두 결과를 `UNION ALL`로 합쳐서 보여준다.

오늘 데이터가 왜 적을까?
전체 데이터가 1억 건이어도, 오늘 하루치 주문은 몇 만 건, 많아야 몇십만 건 수준일 것이다. 전체를 뒤지는 것보다 오늘 하루치만 인덱스를 타고 집계하는 것은 DB가 충분히 감당할 수 있다. 물론 반드시 적절한 인덱스가 있어야 한다.

"이 전략이 유효하려면 반드시 원본 테이블의 날짜 컬럼에 인덱스가 있어야 한다"
만약 `orders` 테이블에 전체 데이터가 1억 건이 쌓여 있는데 `order_date`에 인덱스가 없다면 어떻게 될까? 데이터베이스는 오늘 데이터를 찾기 위해 1억 건을 처음부터 끝까지 다 뒤져야 한다. 결과적으로 **풀 테이블 스캔(Full Table Scan)**이 발생하며 서비스에 점점 심각한 문제가 발생할 것이다.

실습: 하이브리드 쿼리 작성

상황을 가정해보자. 현재 시점은 **2026년 1월 4일 점심시간**이다.
우리는 1월 1일부터 1월 3일까지의 데이터는 통계 테이블에서, 그리고 오늘(1월 4일) 쌓이고 있는 데이터는 원본 주문 테이블에서 가져와서 합쳐야 한다.

- 1월 1일 ~ 1월 3일: 통계 테이블(`daily_sales_stats`)에 이미 저장되어 있음.
- 1월 4일: 원본 테이블(`orders`)에만 있고 통계 테이블에는 없음.

이를 위해 실습 환경을 깨끗하게 다시 세팅해보자. 기존 테이블을 삭제하고 이 시나리오에 맞는 데이터를 입력한다.

```
-- 1. 테이블 초기화 (기존 테이블 삭제 후 재생성)
DROP TABLE IF EXISTS daily_sales_stats;
DROP TABLE IF EXISTS orders;

-- 통계 테이블 생성
CREATE TABLE daily_sales_stats (
    stat_date DATE NOT NULL,
    total_order_count INT NOT NULL DEFAULT 0,
```

```
    total_sales_amount BIGINT NOT NULL DEFAULT 0,
    PRIMARY KEY (stat_date)
);

-- 주문 테이블 생성
CREATE TABLE orders (
    order_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    customer_id BIGINT NOT NULL,
    total_amount INT NOT NULL,
    order_status VARCHAR(20) NOT NULL,
    order_date DATETIME NOT NULL
);

CREATE INDEX idx_orders_order_date ON orders (order_date);

-- 2. 과거 데이터 입력 (통계 테이블)
-- 1월 1일 ~ 1월 3일 데이터는 이미 배치가 돌아서 통계 테이블에 요약되어 있다고 가정한다.
INSERT INTO daily_sales_stats (stat_date, total_order_count,
total_sales_amount) VALUES
('2026-01-01', 3, 50000),
('2026-01-02', 2, 80000),
('2026-01-03', 1, 45000);

-- 3. 오늘 실시간 데이터 입력 (주문 테이블)
-- 1월 4일(오늘) 주문은 아직 배치가 돌지 않았으므로 원본 orders 테이블에만 존재한다.
INSERT INTO orders (customer_id, total_amount, order_status, order_date)
VALUES
(10, 60000, 'COMPLETED', '2026-01-04 10:00:00'), -- 오늘 오전 주문 1
(11, 20000, 'COMPLETED', '2026-01-04 11:30:00'); -- 오늘 오전 주문 2
```

데이터 준비가 끝났다. 이제 **2026년 1월 1일부터 오늘(1월 4일) 현재까지의 일별 매출**을 한 번에 조회하는 쿼리를 만들어보자. `UNION ALL`을 사용하여 두 테이블의 결과를 합칠 것이다.

```
-- 1. 과거 데이터 (통계 테이블 조회)
SELECT
    stat_date,
    total_order_count,
    total_sales_amount
FROM daily_sales_stats
WHERE stat_date >= '2026-01-01' AND stat_date < '2026-01-04'
```

```
UNION ALL

-- 2. 오늘 실시간 데이터 (원본 테이블 직접 집계)
SELECT
    DATE(order_date) as stat_date,
    COUNT(*) as total_order_count,
    SUM(total_amount) as total_sales_amount
FROM orders
WHERE order_date >= '2026-01-04 00:00:00'
  AND order_status = 'COMPLETED'
GROUP BY DATE(order_date);
```

[실행 결과]

stat_date	total_order_count	total_sales_amount
2026-01-01	3	50000
2026-01-02	2	80000
2026-01-03	1	45000
2026-01-04	2	80000

하이브리드 통계를 활용한 월간 합계

방금 우리는 `UNION ALL`을 통해 과거 데이터와 실시간 데이터를 합쳐서 리스트 형태로 출력했다. 이번에는 이 데이터를 기반으로 **'2026년 1월'이라는 월 정보와 함께 총매출 합계**를 구해보자.

조회하는 것과 원리는 똑같다. `UNION ALL`로 합친 결과를 마치 하나의 테이블인 것처럼 감싸서(Subquery), 그 위에서 `GROUP BY`를 수행하면 된다.

```
SELECT
    DATE_FORMAT(stat_date, '%Y-%m') as stat_month,
    SUM(total_order_count) AS monthly_order_count,
    SUM(total_sales_amount) AS monthly_sales_amount
FROM (
    -- 1. 과거 데이터 (통계 테이블)
```

```
    SELECT
        stat_date,
        total_order_count,
        total_sales_amount
    FROM daily_sales_stats
    WHERE stat_date >= '2026-01-01' AND stat_date < '2026-01-04'

    UNION ALL

    -- 2. 오늘 실시간 데이터 (원본 테이블)
    SELECT
        DATE(order_date) as stat_date,
        COUNT(*) as total_order_count,
        SUM(total_amount) as total_sales_amount
    FROM orders
    WHERE order_date >= '2026-01-04 00:00:00'
      AND order_status = 'COMPLETED'
    GROUP BY DATE(order_date)
) AS hybrid_table
GROUP BY DATE_FORMAT(stat_date, '%Y-%m');
```

[실행 결과]

stat_month	monthly_order_count	monthly_sales_amount
2026-01	8	255000

이것이 바로 실전 설계다

결과를 보라. 1일~3일 데이터(175,000원)는 가벼운 통계 테이블에서 가져왔고,4일 데이터(80,000원)만 실시간으로 계산해서 합쳐 정확히 255,000원이 나왔다.

이 쿼리가 수억 건의 데이터 속에서도 빠르게 동작하는 이유는 역할 분담이 확실하기 때문이다.

- **통계 테이블**: 이미 계산이 끝난 3개의 행만 읽는다. (부하 거의 없음)
- **원본 테이블**: `order_date`에 인덱스가 있다면, 오늘 생성된 소량의 데이터만 스캔한다. (부하 매우 적음)

결국 데이터베이스는 수억 건을 뒤지는 대신 수십~수백 건 정도의 데이터만 핸들링하게 된다. 이것이 바로 대규모 서비스를 지탱하는 통계 설계의 핵심 원리다.

사용자는 이것이 내부적으로 어떻게 동작하는지 알 필요가 없다. 그저 "와, 실시간으로 이번 달 매출이 집계되면서 속도도 빠르네요!"라고 할 것이다.

이 방식의 핵심은 **오늘 하루치 데이터는 DB가 충분히 실시간으로 집계할 수 있다**는 점을 이용하는 것이다. 인덱스만 잘 걸려 있다면(`order_date`), 오늘 하루치 데이터를 집계하는 것은 부하가 크지 않다.

이렇게 **통계 테이블(과거)**과 **원본 테이블(현재)**을 적절히 섞어 쓰는 하이브리드 전략을 사용하면, 대용량 데이터 환경에서도 성능과 실시간성이라는 두 마리 토끼를 모두 잡을 수 있다.

멱등성 설계

지난 시간에 우리는 원본 테이블과 분리된 **통계 테이블**을 만들었고, 실시간 데이터는 **하이브리드 방식**으로 조회하여 성능과 편의성이라는 두 마리 토끼를 잡았다.

하지만 실무에서 통계 시스템을 운영하다 보면 정말 끔찍한 일들이 벌어진다.
"어? 어제 배치가 두 번 돌았나 봐요. 매출이 두 배로 뻥튀기 됐어요!"
"3일 전 주문이 오늘 취소됐는데, 3일 전 통계에는 반영이 안 됐는데요?"

이런 문제는 왜 발생할까? 바로 **멱등성(Idempotency)**을 고려하지 않고 배치를 짰기 때문이다. 오늘은 통계 데이터를 다룰 때 가장 중요한 원칙인 멱등성 설계에 대해 깊이 있게 다뤄보자.

잘못된 설계: 누적 업데이트 방식의 함정

초보 개발자가 가장 흔히 저지르는 실수가 있다. 통계 데이터를 갱신할 때 `UPDATE` 문을 사용해서 기존 값에 더하기를 하는 것이다. 이를 '증분 업데이트(Incremental Update)'라고 하는데, 통계 배치에서는 매우 위험한 방식이다.

왜 위험한지 직접 눈으로 확인해보자.

상황 설정: 1월 1일의 추가 매출 집계
시스템에 1월 1일자 통계 데이터가 초기화되어 있다고 가정하자. 여기에 배치가 돌면서 매출액 50,000원을 더하는 상황이다.

```sql
-- 1. 테이블 초기화 및 0원 세팅
DROP TABLE IF EXISTS daily_sales_stats;

CREATE TABLE daily_sales_stats (
    stat_date DATE NOT NULL,
    total_order_count INT NOT NULL DEFAULT 0,
    total_sales_amount BIGINT NOT NULL DEFAULT 0,
    PRIMARY KEY (stat_date)
);

INSERT INTO daily_sales_stats (stat_date, total_order_count,
total_sales_amount) VALUES ('2026-01-01', 0, 0);
```

- 데이터가 초기화 되어 있고, 처음에는 0원이 들어가있다. 이 값을 계속 더하기 하는 방식으로 배치를 만들 예정이다.

잘못된 배치 실행 시나리오

이제 개발자가 작성한 '누적 업데이트' 쿼리를 실행해보자.

```sql
-- [배치 실행 1회차] 정상적인 상황
-- 1월 1일 매출 50,000원을 집계하여 더한다.
UPDATE daily_sales_stats
SET total_sales_amount = total_sales_amount + 50000,
    total_order_count = total_order_count + 1
WHERE stat_date = '2026-01-01';

-- 결과 확인
SELECT * FROM daily_sales_stats;
```

[실행 결과 1]

stat_date	total_order_count	total_sales_amount
2026-01-01	1	50000

여기까지는 문제가 없다. 0원에 50,000원을 더해서 50,000원이 되었다.

그런데, 배치가 실행되다가 네트워크 오류로 타임아웃이 발생했다고 가정해보자. 개발자나 운영자는 "어? 배치가 실패했네? 다시 돌려야겠다."라고 생각하고 **배치를 재실행**한다. 이것이 비극의 시작이다.

```
-- [배치 실행 2회차] 재실행 상황 (사고 발생)
-- 배치를 다시 실행하면 기존 값(50000)에 또 50,000원을 더해버린다.
UPDATE daily_sales_stats
SET total_sales_amount = total_sales_amount + 50000,
    total_order_count = total_order_count + 1
WHERE stat_date = '2026-01-01';

-- 결과 확인
SELECT * FROM daily_sales_stats;
```

[실행 결과 2]

stat_date	total_order_count	total_sales_amount
2026-01-01	2	100000

결과를 보라. 실제 매출은 50,000원인데, 통계 테이블에는 100,000원이 찍혀있다. 매출이 두 배로 뻥튀기된 것이다.

이런 방식을 **비멱등적(Non-idempotent)**이라고 한다.

- **멱등성(Idempotency)**이란, 연산을 한 번 실행하든, 100번 실행하든 **결과가 항상 같아야 한다**는 성질이다.
- `total_sales_amount = total_sales_amount + 50000`과 같은 로직은 실행할 때마다 값이 변하므로 멱등성이 깨진 것이다.
- 실행 횟수에 따라 결과가 달라지기 때문에 언제든지 데이터가 오염될 수 있다.

배치 프로그램이나 통계 쿼리는 언제든 실패할 수 있고, 언제든 재실행될 수 있어야 한다. 따라서 **통계 집계 로직은 반드시 멱등성을 보장하도록 설계해야 한다.**

올바른 설계: 멱등성(Idempotency)이란?

멱등성(Idempotency)이란, 연산을 한 번 실행하든, 백 번 실행하든 **그 결과가 항상 같아야 한다**는 성질이다.

통계 배치 시스템에서 멱등성을 지키는 가장 단순하고 확실한 방법은 **"지우고 다시 쓰기(DELETE & INSERT)"** 전략

이다.
"기존에 값이 있든 없든, 틀렸든 맞든, 그냥 싹 지우고 원본 데이터를 기반으로 다시 계산해서 넣는다."

이 전략이 있으면 우리는 두려움이 사라진다.

- "배치가 실패했나요? 다시 돌리세요."
- "데이터가 꼬였나요? 다시 돌리세요."
- "3일 전 데이터가 변경됐나요? 그 날짜 배치를 다시 돌리세요."

모든 문제 해결 방법이 **"재실행"** 하나로 통일된다. 심플함이 최고의 미덕이다.

실습: 멱등한 배치 프로세스 구현

우리는 앞서 만든 `daily_sales_stats` 테이블을 그대로 사용한다.
2026년 1월 1일의 통계를 다시 계산해야 하는 상황을 가정해보자.

예제를 위해 먼저 다음과 같이 1월 1일 주문 테이블과 데이터를 다시 만들자.

```
DROP TABLE IF EXISTS orders;

-- 주문 테이블 생성
CREATE TABLE orders (
    order_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    customer_id BIGINT NOT NULL,
    total_amount INT NOT NULL,
    order_status VARCHAR(20) NOT NULL,
    order_date DATETIME NOT NULL
);

INSERT INTO orders (customer_id, total_amount, order_status, order_date)
VALUES
(1, 10000, 'COMPLETED', '2026-01-01 10:00:00'),
(2, 25000, 'COMPLETED', '2026-01-01 14:30:00'),
(3, 15000, 'COMPLETED', '2026-01-01 18:20:00')
```

이 데이터를 기반으로 통계를 만들어보자.

1단계: 해당 날짜의 통계 데이터 삭제

먼저 해당 날짜(`2026-01-01`)에 이미 저장된 통계 데이터가 있다면 무조건 삭제한다. 데이터가 없어도 상관없다.

```
DELETE FROM daily_sales_stats WHERE stat_date = '2026-01-01';
```

[실행 결과]

```
1 row affected
```

2단계: 해당 날짜의 통계 데이터 생성 (INSERT)

깨끗해진 상태에서 원본 `orders` 테이블을 읽어 다시 집계해 넣는다.

```
INSERT INTO daily_sales_stats (stat_date, total_order_count,
total_sales_amount)
SELECT
    DATE(order_date),
    COUNT(*),
    SUM(total_amount)
FROM orders
WHERE order_date >= '2026-01-01 00:00:00'
  AND order_date < '2026-01-02 00:00:00'
  AND order_status = 'COMPLETED'
GROUP BY DATE(order_date);
```

```
SELECT * FROM daily_sales_stats;
```

[실행 결과]

stat_date	total_order_count	total_sales_amount
2026-01-01	3	50000

이 1단계와 2단계를 하나의 트랜잭션으로 묶어서 실행하면 완벽한 멱등성이 보장된다. 이 쿼리 세트는 하루에 100번을

실행해도 결과는 항상 `orders` 테이블의 현재 상태와 정확히 일치한다.

다시 실행해보자.

```
-- 1. 삭제
DELETE FROM daily_sales_stats WHERE stat_date = '2026-01-01';

-- 2. 재생성
INSERT INTO daily_sales_stats (stat_date, total_order_count,
total_sales_amount)
SELECT
    DATE(order_date),
    COUNT(*),
    SUM(total_amount)
FROM orders
WHERE order_date >= '2026-01-01 00:00:00'
  AND order_date < '2026-01-02 00:00:00'
  AND order_status = 'COMPLETED'
GROUP BY DATE(order_date);
```

```
SELECT * FROM daily_sales_stats;
```

[실행 결과]

stat_date	total_order_count	total_sales_amount
2026-01-01	3	50000

여러번 반복 실행해도 기존과 같은 결과를 확인할 수 있다.

과거 데이터 변경 대응 (환불 발생 시나리오)

쇼핑몰에서 가장 빈번한 이슈는 **과거 주문의 취소/환불**이다.

1월 1일에 주문하고 결제까지 완료되어 매출로 잡혔는데, 1월 4일에 고객이 변심하여 환불을 요청했다.

이때 통계 시스템은 어떻게 동작해야 할까?

상황 재현

먼저 1월 1일에 주문했던 `order_id = 1`번 주문(10,000원)을 취소 처리해보자.

```
-- 1월 4일에 1월 1일자 주문을 취소함
UPDATE orders
SET order_status = 'CANCELLED'
WHERE order_id = 1;
```

[실행 결과]

```
Query OK, 1 row affected
```

문제 확인

현재 `daily_sales_stats` 테이블의 1월 1일 통계는 과거에 계산된 값이므로, 취소된 10,000원이 여전히 매출로 잡혀있다.

```
SELECT * FROM daily_sales_stats WHERE stat_date = '2026-01-01';
```

[실행 결과]

stat_date	total_order_count	total_sales_amount
2026-01-01	3	50000

- 원래 50,000원이었는데, 10,000원이 취소되었으니 40,000원이 되어야 한다.

해결: 배치 재실행

우리의 설계는 멱등하다. 고민할 것 없이 1월 1일자 배치를 다시 돌리면 된다. (DELETE ➜ INSERT)

```
-- 1. 삭제
DELETE FROM daily_sales_stats WHERE stat_date = '2026-01-01';

-- 2. 재생성
```

```sql
INSERT INTO daily_sales_stats (stat_date, total_order_count,
total_sales_amount)
SELECT
    DATE(order_date),
    COUNT(*),
    SUM(total_amount)
FROM orders
WHERE order_date >= '2026-01-01 00:00:00'
  AND order_date < '2026-01-02 00:00:00'
  AND order_status = 'COMPLETED'
GROUP BY DATE(order_date);
```

[실행 결과]

```
Query OK, 1 row affected
Query OK, 1 row affected
```

다시 조회해보자.

```sql
SELECT * FROM daily_sales_stats WHERE stat_date = '2026-01-01';
```

[실행 결과]

stat_date	total_order_count	total_sales_amount
2026-01-01	2	40000

정확하게 40,000원으로 갱신되었다. (`order_count`도 3건에서 2건으로 줄었다.)

이것이 바로 **멱등성 설계의 힘**이다. 복잡한 수정 로직(`UPDATE ... -10000`)을 짤 필요 없이, 언제든 원본 상태에 맞춰 통계를 "동기화"할 수 있다.

실무에서 배치는 반드시 멱등하게 설계해야 한다.

마이크로 배치

지난 시간에 우리는 '통계 테이블'과 '원본 테이블'을 함께 조회하는 **하이브리드 방식**으로 실시간 매출을 구현했다. 데이터 양이 적당할 때는 이 방식이 베스트다.

하지만 우리 쇼핑몰이 '초대박'이 나서 하루 주문량이 100만 건, 1,000만 건을 넘어서면 어떻게 될까? 그리고 수 많은 팀에서 실시간 통계 정보를 조회한다고 가정해보자. 아무리 인덱스를 잘 타도, 사용자가 조회 버튼을 누를 때마다 수백만 건의 오늘 데이터를 집계(`SUM`, `COUNT`)하는 것은 DB에 큰 부담이다.

이때 우리는 기술적인 타협을 해야 한다.
"대표님, **완전 실시간** 대신 **5분 전 데이터**까지만 보여주는 건 어떨까요? 대신 조회 속도는 0.1초입니다."

이것이 바로 **마이크로 배치(Micro-batch)** 전략이다.
이번에는 마이크로 배치를 사용해서 실시간성과 성능의 조화를 이루어보자.

마이크로 배치 설계

핵심은 **'오늘 하루치 통계'를 위한 임시 테이블**을 하나 더 만드는 것이다. 그리고 이 테이블을 아주 짧은 주기(예: 1분, 5분)로 갱신한다. 이름 그대로 아주 작고 가벼운 배치를 만드는 것이다.

1. 당일 통계용 테이블 생성

기존의 `daily_sales_stats`는 어제까지의 '확정된' 데이터만 저장한다. 오늘의 데이터를 저장할 `intraday_sales_stats` (장중 통계) 테이블을 별도로 만든다.

```
DROP TABLE IF EXISTS intraday_sales_stats;

CREATE TABLE intraday_sales_stats (
    today_date DATE NOT NULL,
    total_order_count INT NOT NULL DEFAULT 0,
    total_sales_amount BIGINT NOT NULL DEFAULT 0,
    updated_at DATETIME,
    PRIMARY KEY (today_date)
);
```

2. 마이크로 배치 실행 (삭제 후 재생성 전략)

이제 5분마다 실행되는 스케줄러가 있다고 가정하자.

가장 먼저 떠오르는 방법은 앞서 배운 **"지우고 다시 쓰기(DELETE & INSERT)"** 전략이다. 멱등성이 보장되는 아주 확실한 방법이기 때문이다.

먼저 실습을 위해 오늘이 **2026년 1월 2일**이라고 가정하고, `orders` 테이블에 오늘자 주문 데이터를 몇 건 입력해보자.

```
-- 실습용 데이터 삽입 (오늘이 2026-01-02라고 가정)
INSERT INTO orders (customer_id, total_amount, order_status, order_date)
VALUES
(1, 50000, 'COMPLETED', '2026-01-02 10:00:00'),
(2, 30000, 'COMPLETED', '2026-01-02 11:30:00'),
(3, 120000, 'COMPLETED', '2026-01-02 12:15:00');
```

이제 이 데이터를 바탕으로 마이크로 배치를 실행해보자. 오늘 날짜를 기준으로 기존 데이터를 삭제하고 다시 집계하여 입력하는 과정이다.

```
-- 1. 트랜잭션 시작
START TRANSACTION;

-- 2. 기존 오늘자(2026-01-02) 통계 삭제
DELETE FROM intraday_sales_stats WHERE today_date = '2026-01-02';

-- 3. 오늘자 통계 다시 계산해서 입력
INSERT INTO intraday_sales_stats (today_date, total_order_count,
total_sales_amount, updated_at)
SELECT
    DATE(order_date),
    COUNT(*),
    SUM(total_amount),
    '2026-01-02 13:05:00' -- NOW()를 사용해야 하지만, 예시를 위해 시간 지정
FROM orders
WHERE order_date >= '2026-01-02' AND order_date < '2026-01-03' -- 오늘 데이터 선택
  AND order_status = 'COMPLETED'
GROUP BY DATE(order_date);

-- 4. 커밋
COMMIT;
```

트랜잭션을 사용했기 때문에 `DELETE`가 실행된 직후 `INSERT`가 완료되기 전까지의 짧은 순간에도, 다른 사용자는 삭제된 상태를 볼 수 없다. 즉, "데이터가 비어 보이는 문제"는 없다.

확인해보자.

```
SELECT * FROM intraday_sales_stats;
```

[실행 결과]

today_date	total_order_count	total_sales_amount	updated_at
2026-01-02	3	200000	2026-01-02 13:05:00

- 여기서 핵심은 `updated_at` 컬럼이다. 이 컬럼의 값을 통해 사용자는 배치가 실행된 최종 시간을 알 수 있다.

3. 조회 쿼리 변경

이제 조회 쿼리는 원본 `orders` 테이블을 전혀 건드리지 않는다.

과거 통계(`daily_sales_stats`)와 **오늘 통계(`intraday_sales_stats`)** 두 개의 작은 테이블만 합치면 된다.

```
SELECT
    stat_date,
    total_order_count,
    total_sales_amount
FROM daily_sales_stats
WHERE stat_date >= '2026-01-01' -- 조회 기간 시작

UNION ALL

SELECT
    today_date as stat_date,
    total_order_count,
    total_sales_amount
FROM intraday_sales_stats
WHERE today_date = '2026-01-02'; -- 오늘 데이터
```

[실행 결과]

stat_date	total_order_count	total_sales_amount
2026-01-01	2	40000
2026-01-02	3	200000

- 통계로 만들어진 1월 1일 데이터와, 실시간으로 만들어진 1월 2일 데이터를 함께 조회할 수 있다.

장점과 단점

장점

- **극강의 조회 성능**: 원본 데이터가 수 백만 건이어도 상관없다. 조회 쿼리는 오늘 날짜로 단 1건의 데이터를 읽는다.
- **DB 보호**: 수많은 관리자가 동시에 새로고침을 눌러도, 무거운 집계 쿼리는 5분에 한 번만 실행된다.

단점

- **실시간성 지연**: 사용자는 최대 5분 전의 데이터를 본다. (보통 비즈니스적으로는 "기준 시간: 14:05"이라고 표시해주면 문제없다.)
- **구현 복잡도**: 스케줄러와 별도의 테이블을 관리해야 한다.

삭제 후 재생성 방식의 숨겨진 비효율

하지만 빈번하게 실행되는 마이크로 배치에서 `DELETE` 후 `INSERT` 방식을 사용하는 것은 **성능적으로 비효율적**이다.

1. **불필요한 리소스 낭비**
   - 기존 데이터를 `DELETE` 하면 DB는 복구를 위해 **Undo Log**를 생성한다.
   - 다시 `INSERT` 하면 **Redo Log**를 기록하고 **인덱스를 다시 구성**한다.
   - 사실상 값만 조금 바뀌었을 뿐인데, 멀쩡한 데이터를 지우고 다시 만드는 과정에서 DB 엔진이 불필요한 일을 많이 하게 된다.

2. **ID 고갈 및 인덱스 파편화**
   - 만약 PK가 `AUTO_INCREMENT`라면, 5분마다 데이터가 지워지고 새로 생기면서 ID 값이 빠르게 증가하게 된다. (이번 예제에서는 PK가 AUTO_INCREMENT가 아니므로 상관없다.)
   - 잦은 삭제와 삽입은 데이터 페이지의 파편화를 유발할 수 있다.

결과적으로 너무 자주 삭제하고 다시 입력하는 방식은 데이터베이스 성능에 좋지 않은 영향을 준다는 점이다.
하루에 한 번 도는 새벽 배치라면 상관없지만, 1분, 5분 단위로 도는 배치에서 굳이 멀쩡한 행(Row)을 파괴하고 다시 만들 필요는 없다.

우리는 데이터를 지우지 않으면서, 자연스럽게 **값만 갱신**하는 더 우아한 방법이 필요하다.

UPSERT 최적화

앞서 설명한 문제를 최적화 하는 방법은 5분마다 값을 DELETE하고 다시 INSERT 하는 대신에 단순히 값을 UPDATE 하는 것이다.
그런데 아직 당일 통계 데이터가 입력되지 않았다면 오늘 행 자체가 없다. 따라서 이 경우에는 INSERT를 해서 먼저 당일 행을 만들어야 한다. 이후에는 UPDATE를 하면 된다. 이렇게 하면 앞서 설명한 불필요한 리소스 낭비가 줄어든다.

조회 후 등록 수정 (Select-Insert-Update) 패턴

무작정 지우고 다시 쓰는 방식은 데이터베이스 입장에서 효율이 떨어진다.
가장 일반적인 방법은 **"데이터가 있는지 먼저 확인해보고(`SELECT`), 없으면 넣고(`INSERT`), 있으면 수정(`UPDATE`) 하는 것"**이다.

이 로직은 보통 자바나 파이썬 같은 애플리케이션 코드 레벨에서 제어하지만, 우리는 데이터베이스 관점에서 어떤 SQL들이 실행되는지 단계별로 살펴보자.

예제를 진행하기 위해 당일 통계용 테이블을 삭제하고 다시 만들자

```
DROP TABLE IF EXISTS intraday_sales_stats;

CREATE TABLE intraday_sales_stats (
    today_date DATE NOT NULL,
    total_order_count INT NOT NULL DEFAULT 0,
    total_sales_amount BIGINT NOT NULL DEFAULT 0,
    updated_at DATETIME,
    PRIMARY KEY (today_date)
);
```

1단계: 데이터 존재 여부 확인 (SELECT)

먼저 오늘 날짜(`2026-01-02`)에 해당하는 통계 데이터가 이미 존재하는지 확인해야 한다.

```
SELECT count(*)
FROM intraday_sales_stats
WHERE today_date = '2026-01-02';
```

[실행 결과]

count(*)
0

아직 오늘 날짜 (`2026-01-02`)의 통계를 만든 적이 없기 때문에 행이 존재하지 않는다.

(앞서 테이블을 재생성했기 때문에, 아직 오늘 날짜(2026-01-02)의 통계 데이터는 존재하지 않는다.)

결과가 `1`이라면 이미 데이터가 있다는 뜻이고, `0`이라면 없다는 뜻이다. 이 결과에 따라 다음 행동이 갈린다.

2단계: 상황별 쿼리 실행

상황 A. 데이터가 없는 경우 (New)

만약 오늘 00시 05분에 첫 배치가 돌았다면, 아직 `2026-01-02` 행이 없을 것이다. 이때는 `INSERT`를 수행한다.

```
INSERT INTO intraday_sales_stats (today_date, total_order_count,
total_sales_amount, updated_at)
SELECT
    DATE(order_date),
    COUNT(*),
    SUM(total_amount),
    '2026-01-02 00:05:00' -- NOW()를 사용해야 하지만, 예시에서는 직접 입력
FROM orders
WHERE order_date >= '2026-01-02' AND order_date < '2026-01-03'
  AND order_status = 'COMPLETED'
GROUP BY DATE(order_date);
```

상황 B. 데이터가 이미 있는 경우 (Update)

다시 오늘 날짜(`2026-01-02`)에 해당하는 통계 데이터가 이미 존재하는지 확인해보자.

```
SELECT count(*)
FROM intraday_sales_stats
WHERE today_date = '2026-01-02';
```

[실행 결과]

count(*)
1

앞서 아직 오늘 날짜 (`2026-01-02`)의 통계를 만들었기 때문에 행이 존재한다.

앞선 `SELECT` 결과가 `1`이라면, 이미 행이 존재하므로 `UPDATE` 문을 실행해 값만 바꿔치기한다. 이 방식은 행(Row)을 삭제하지 않으므로 인덱스를 재구성하거나 ID가 낭비되는 일이 없다.

```
-- 애플리케이션에서 미리 계산된 값을 파라미터로 바인딩한다고 가정한다.
-- 여기서는 이해를 돕기 위해 서브쿼리를 사용해 값을 갱신한다.
UPDATE intraday_sales_stats
SET
    total_order_count = (SELECT COUNT(*) FROM orders WHERE order_date >=
'2026-01-02' AND order_date < '2026-01-03' AND order_status = 'COMPLETED'),
    total_sales_amount = (SELECT SUM(total_amount) FROM orders WHERE
order_date >= '2026-01-02' AND order_date < '2026-01-03' AND order_status =
'COMPLETED'),
    updated_at = '2026-01-02 00:10:00' -- NOW()를 사용해야 하지만, 예시에서는 직접 입력
WHERE today_date = '2026-01-02';
```

- UPDATE를 사용했지만 이 방식은 멱등하다. 기존 값을 증가하는 것이 아니라 기존 값을 모두 덮어써 버리기 때문이다.
- 배치를 여러번 실행해도 원본이 같다면 결과는 항상 같다.

확인해보자. `updated_at` 시간만 변경되고 데이터는 안전하게 갱신되었을 것이다.

```
SELECT * FROM intraday_sales_stats;
```

[실행 결과]

today_date	total_order_count	total_sales_amount	updated_at
2026-01-02	3	200000	2026-01-02 00:10:00

SELECT → INSERT/UPDATE 방식 정리

이 방식은 불필요한 삭제를 막아주지만, 쿼리를 **두 번(SELECT → INSERT/UPDATE)** 실행해야하는 단점이 있다.
네트워크 통신 비용이 두 배로 드는 셈이다.
이쯤 되면 여러분은 이런 생각이 들 것이다.

"데이터베이스한테 **'알아서 확인하고 없으면 넣고 있으면 수정해줘'**라고 한 번에 시킬 수는 없을까?"

물론 있다. MySQL이 제공하는 강력한 기능인 `UPSERT` (INSERT + UPDATE) 구문에 대해 알아보자.

고급 기법: ON DUPLICATE KEY UPDATE 활용

MySQL에서는 `DELETE` 후 `INSERT` 하는 과정의 비효율을 해결해주는 기능을 제공한다. 바로 `INSERT ... ON DUPLICATE KEY UPDATE` 구문이다. 이를 흔히 **UPSERT** (Update + Insert)라고 부른다.

ON DUPLICATE KEY UPDATE 문법

기본적인 문법 구조는 `INSERT` 문 뒤에 `ON DUPLICATE KEY UPDATE` 절을 붙이는 형태다.

```
INSERT INTO 테이블명 (컬럼1, 컬럼2, ...)
VALUES (값1, 값2, ...)
ON DUPLICATE KEY UPDATE
  컬럼1 = 새로운_값,
  컬럼2 = 새로운_값;
```

- `INSERT INTO ...`: 데이터를 삽입하는 일반적인 쿼리와 같다. 데이터베이스는 가장 먼저 이 데이터를 입력을 시도한다.
- `ON DUPLICATE KEY`: 만약 삽입하려는 데이터의 PK(또는 Unique Key)가 이미 테이블에 존재해서 충돌이 발생한다면?
- `UPDATE ...`: 오류를 발생시키는 대신, 뒤에 작성된 `UPDATE` 로직을 실행하여 기존 행의 값을 수정한다.

이 구문을 사용하면 "일단 넣어보고, 안 되면 수정해라"라는 로직을 SQL 한 문장으로 깔끔하게 처리할 수 있다.

UPSERT의 원리

UPSERT는 이름 그대로 **"없으면 넣고(Insert), 있으면 고친다(Update)"**는 뜻이다. 작동 원리는 아주 직관적이다.

1. **INSERT 시도**: 데이터베이스는 먼저 데이터를 `INSERT` 하려고 시도한다.
2. **충돌 확인**: 이때, `PRIMARY KEY`나 `UNIQUE KEY`가 이미 존재하는지 확인한다.
    - **중복이 없다면**: 정상적으로 `INSERT`가 실행된다. (새로운 데이터 생성)
    - **중복이 있다면**: `INSERT`를 포기하고, `UPDATE` 문을 실행한다. (기존 데이터 갱신)

이 방식은 기존 행(Row)을 삭제하지 않고 필요한 컬럼의 값만 쓱 바꾸기 때문에 `DELETE` + `INSERT` 방식보다 훨씬 효율적이고 가볍다.

예제를 진행하기 위해 당일 통계용 테이블을 삭제하고 다시 만들자

```
DROP TABLE IF EXISTS intraday_sales_stats;

CREATE TABLE intraday_sales_stats (
    today_date DATE NOT NULL,
    total_order_count INT NOT NULL DEFAULT 0,
    total_sales_amount BIGINT NOT NULL DEFAULT 0,
    updated_at DATETIME,
    PRIMARY KEY (today_date)
);
```

아직 오늘 날짜(`2026-01-02`)의 통계를 만든 적이 없기 때문에 행이 존재하지 않는다.

UPSERT 적용

마이크로 배치를 UPSERT 방식으로 변경해보자. `today_date`가 PK이므로 날짜가 같으면 자동으로 `UPDATE`로 전환된다.
UPSERT의 사용 방식이 MySQL 최신 버전에서는 약간 달라졌기 때문에 다음 내용을 확인하자.

MySQL 8.0.20+ 이전

```
INSERT INTO intraday_sales_stats (today_date, total_order_count,
total_sales_amount, updated_at)
```

```
SELECT
    DATE(order_date),
    COUNT(*),
    SUM(total_amount),
    NOW()
FROM orders
WHERE order_date >= '2026-01-02'
  AND order_status = 'COMPLETED'
GROUP BY DATE(order_date)
ON DUPLICATE KEY UPDATE
    total_order_count = VALUES(total_order_count),    -- 입력하려고 했던 바로 그 값
(New)
    total_sales_amount = VALUES(total_sales_amount), -- 입력하려고 했던 바로 그 값
(New)
    updated_at = VALUES(updated_at);                  -- 입력하려고 했던 바로 그 값
(New)
```

- 아직 오늘 날짜 (`2026-01-02`)의 통계를 만든 적이 없기 때문에 행이 존재하지 않는다.
- 이 경우 데이터가 없기 때문에 INSERT 가 발생한다.

> ✨ **MySQL 8.0.20+ VALUES()**
> 과거에는 컬럼의 값과 새로운 값을 구분하기 위해 VALUES() 함수를 사용했다.
> 이 함수는 MySQL 8.0.20+ 부터 Deprecated(권장하지 않음)되어서 권장하지 않는다.

MySQL 8.0.20+ 이후

```
INSERT INTO intraday_sales_stats (today_date, total_order_count,
total_sales_amount, updated_at)
SELECT * FROM (
    -- 1. 먼저 오늘치 데이터를 집계하는 서브쿼리를 만든다.
    SELECT
        DATE(order_date) as today_date,
        COUNT(*) as total_order_count,
        SUM(total_amount) as total_sales_amount,
        NOW() as updated_at
    FROM orders
    WHERE order_date >= '2026-01-02'
      AND order_status = 'COMPLETED'
    GROUP BY DATE(order_date)
) AS new_stats -- 2. 이 집계 결과(테이블)에 'new_stats'라는 별칭을 붙인다.
ON DUPLICATE KEY UPDATE
```

```
    total_order_count = new_stats.total_order_count,    -- 서브쿼리의 값으로 덮어쓰기
    total_sales_amount = new_stats.total_sales_amount, -- 서브쿼리의 값으로 덮어쓰기
    updated_at = new_stats.updated_at;                  -- 서브쿼리의 시간으로 갱신
```

- 최신 방식에서는 컬럼의 값과 새로운 값을 구분하기 위해 별칭을 만들어야 하는데, 서브쿼리를 통해 별칭을 만들어야 한다.

```
SELECT * FROM intraday_sales_stats;
```

[실행 결과]

today_date	total_order_count	total_sales_amount	updated_at
2026-01-02	3	200000	2026-01-02 14:11:00

- 실행 결과를 보면 데이터가 INSERT 된 것을 확인할 수 있다.

이번에는 2026-01-02 데이터가 있는 경우 어떻게 되는지 확인하기 위해 같은 쿼리를 한 번 더 실행해보자.

위의 UPSERT 쿼리를 실행하고 결과를 다시 확인해보자.

```
SELECT * FROM intraday_sales_stats;
```

[실행 결과]

today_date	total_order_count	total_sales_amount	updated_at
2026-01-02	3	200000	2026-01-02 14:13:00

- 결과는 같고 `updated_at` 컬럼의 값만 변경된 것을 확인할 수 있다. (`order` 테이블의 데이터가 변경된다면 결과가 달라질 것이다.)
- 지정한 모든 컬럼에 UPDATE가 정상 수행되었다.

이 쿼리는 매우 강력하다. 상세히 뜯어보자.

1. `SELECT`: 먼저 `orders` 테이블에서 오늘치 데이터를 집계한다.
2. `INSERT` **시도**: `intraday_sales_stats`에 오늘 날짜 데이터를 넣으려 한다.
3. **충돌 발생 시**: 이미 오늘 날짜(`today_date`) 데이터가 있으므로 `ON DUPLICATE KEY UPDATE` 절이 실행된다.
    - `VALUES(total_sales_amount)`, 또는 서브쿼리 별칭: 방금 `SELECT`로 집계해온 새로운 값을 의미한다.
    - 결국 기존 값을 새 값으로 안전하게 덮어쓴다. (UPDATE)

UPSERT의 장점

- **원자성(Atomicity)**: `DELETE`와 `INSERT` 사이에 틈이 없다. 기존 데이터를 지웠는데 갑자기 서버가 죽어서 데이터가 날아가는 사고를 막을 수 있다. 이런 부분은 데이터베이스가 처리한다.
- **간결함**: 복잡한 `IF` 로직 없이 하나의 쿼리로 깔끔하게 처리된다.
- **멱등성**: 이 방식이 멱등성을 완벽하게 보장하는 이유는 **'덮어쓰기'** 전략을 취하기 때문이다 100번을 실행해도 결과는 항상 같다. 이것이 우리가 원하던 안전한 설계다.
- **성능**: UPSERT를 사용하는 방식이 DELETE 후 INSERT 방식보다 성능에서 많이 유리하다.

DELETE 후 INSERT 방식과 UPSERT 방식의 성능 비교

지금까지 우리는 데이터를 갱신할 때 무작정 지우고 다시 쓰는(`DELETE` + `INSERT`) 방식이 아니라, `UPSERT`(`INSERT ... ON DUPLICATE KEY UPDATE`)를 사용하는 방법을 배웠다. 이 두 방식이 기능적으로는 "데이터를 최신 상태로 만든다"는 점에서 같아 보일 수 있지만, 데이터베이스 내부에서 일어나는 일과 성능 비용은 매우 크다.

왜 실무에서 데이터를 자주 갱신하는 경우 `DELETE` 후 `INSERT`를 지양하고 `UPSERT`를 권장하는지, 그 성능 차이의 원인과 실제 비용을 정리해보자.

1. 비용 발생의 근본 원인: 인덱스와 로그

데이터베이스에서 데이터를 쓰고 지우는 것은 단순히 공책에 글씨를 썼다 지우는 것보다 훨씬 복잡한 작업을 수반한다.

DELETE + INSERT 방식의 비용 (집을 부수고 다시 짓기)

이 방식은 멀쩡한 집을 허물고(`DELETE`) 그 자리에 똑같은 집을 다시 짓는(`INSERT`) 것과 같다.

1. **인덱스 파괴 및 재구축**: 테이블에 인덱스가 3개 걸려 있다면, `DELETE` 시 3개의 인덱스에서 모두 데이터를 찾아 제거해야 한다. 그리고 `INSERT` 시 다시 3개의 인덱스에 데이터를 끼워 넣어야 한다. 인덱스 정렬을 위한 오버헤드가 두 배로 발생한다.
2. **방대한 로그 생성**: 데이터베이스는 복구를 위해 변경 사항을 모두 기록한다(Redo Log, Undo Log). 데이터를

삭제했다는 기록과 새로 넣었다는 기록을 모두 남겨야 하므로 로그 사용량이 급증한다.

3. **페이지 단편화(Fragmentation)**: 데이터를 지우고 다시 넣다 보면 데이터가 저장된 물리적 공간(페이지)에 구멍이 숭숭 뚫리는 단편화 현상이 발생한다. 이는 나중에 데이터를 조회할 때 성능 저하의 주범이 된다.

UPSERT (UPDATE) 방식의 비용 (인테리어만 바꾸기)

반면 `UPSERT`를 통해 `UPDATE`가 수행되는 것은 집의 골조는 그대로 두고 벽지(`Value`)만 바꾸는 것과 같다.

1. **인덱스 보존**: PK나 인덱스 컬럼을 건드리지 않고 일반 컬럼(예: `total_order_count`)만 수정한다면, 인덱스 트리 구조를 변경할 필요가 전혀 없다. 단순히 값만 덮어쓰면 된다.
2. **최소한의 로그**: 변경된 값에 대한 로그만 남기면 되므로 디스크 I/O가 훨씬 적다.
3. **공간 재사용**: 이미 할당된 공간을 그대로 사용하므로 단편화가 발생하지 않는다.
4. **네트워크 호출 최적화**: 네트워크 호출 횟수가 줄어든다.

2. 실무에서의 실제 성능 차이

그렇다면 실제로 얼마나 차이가 날까?

- **인덱스가 없는 테이블**: 인덱스가 거의 없다면 `DELETE` + `INSERT`도 꽤 빠르다. 하지만 실무에서 인덱스가 없는 주요 테이블은 거의 없다.
- **일반적인 테이블 (인덱스 3~5개)**: 가장 흔한 케이스다. 이 경우 `UPSERT` 방식이 `DELETE` + `INSERT` 방식보다 **최소 2배에서 3배 이상 빠르다.** 인덱스가 많아질수록 이 격차는 기하급수적으로 벌어진다.
- **동시성 처리가 많은 환경**: 쇼핑몰 주문 통계처럼 사용자가 몰리는 환경에서는 성능 차이가 **10배 이상** 벌어지기도 한다. `DELETE` + `INSERT`는 찰나의 순간이지만 데이터가 없는 구간(Gap)을 만들기 때문에, 트랜잭션을 걸어야 할 수 있다. 다른 트랜잭션이 해당 데이터를 조회하거나 잠금을 획득하려 할 때 대기(Lock Wait) 현상을 유발하여 시스템 전체를 느리게 만든다.

결론적으로, 통계 데이터를 자주 갱신하거나 기존 정보를 자주 업데이트할 때는 `UPSERT` 방식을 사용하여 데이터베이스의 부하를 줄일 수 있다.

정리: 언제 무엇을 쓸까?

- **일별/월별 통계 배치 (새벽 실행)**: `DELETE` + `INSERT` 패턴을 추천한다. "지우고 다시 만든다"는 개념이 명확하여 운영 중에 데이터가 꼬였을 때 재처리가 쉽고 디버깅이 편하다. 하루에 한 번 정도의 비효율은 전혀 문제가 되지 않는다.
- **실시간/마이크로 배치 (빈번한 갱신)**: `ON DUPLICATE KEY UPDATE` **(UPSERT)**를 사용해야 한다. 불필요한 삭제와 생성 과정을 줄여 DB의 부하를 최소화할 수 있다. 특히 실시간으로 입력되는 데이터를 기반으로 통계 데이터를 만들어야 한다면 더욱 효과적이다.

실무에서는 **일별 통계 배치는 DELETE + INSERT**, **실시간 집계는 UPSERT**를 주로 사용한다.

[심화] 증분 업데이트 (Incremental Update)

만약 "오늘 하루치 데이터를 다시 읽는 것(`GROUP BY`)"조차 너무 무거워서 5분 안에 안 끝난다면 어떻게 할까?
(예: 하루 주문이 1억 건이라서 집계만 10분이 걸리는 경우)

이때는 **마지막으로 읽은 위치(ID)**를 기억해두고, **새로 추가된 데이터만** 읽어서 더하는 방법이 있다.

1. `batch_meta` 테이블에 `last_order_id`를 저장한다.
2. `SELECT ... FROM orders WHERE order_id > last_order_id` 로 새 데이터만 가져온다.
3. 기존 통계값에 `+` 연산을 하여 업데이트한다. (`UPDATE ... SET count = count + new_count`)
4. `last_order_id`를 갱신한다.

주의: 이 방식은 `UPDATE` 누적 방식이므로 **멱등성**이 깨지기 쉽다. 배치가 중간에 죽거나 중복 실행되면 데이터가 꼬일 위험이 크다. 따라서 정말 극한의 상황이 아니라면 **"오늘 데이터 전체 재집계(UPSERT)"** 방식을 권장한다.

정리

학습 내용 정리

통계 데이터와 성능 문제

- 서비스 초기에는 원본 테이블(`orders`)을 직접 `GROUP BY` 하여 통계를 내도 문제가 없다.
- 데이터가 수천만 건 이상 쌓이면 통계 쿼리가 실행될 때 CPU와 메모리를 과다하게 사용하여 서비스 전체 성능을 저하시킨다.
- 통계 조회로 인해 고객의 주문/결제 트랜잭션이 느려지거나 장애가 발생할 수 있다.
- 대용량 트래픽 환경에서는 통계 처리와 원본 데이터 처리를 분리해야 한다.

통계 테이블 설계

- 통계를 위한 별도의 요약 테이블(Summary Table)을 생성하여 해결한다.
- 배치(Batch) 프로그램을 통해 사용자가 적은 새벽 시간에 원본 데이터를 집계하여 통계 테이블에 저장한다.
- 통계 테이블은 하루에 한 행만 생성되므로 데이터 양이 획기적으로 줄어들어 조회 속도가 매우 빠르다.
- 실시간 데이터를 조회할 수 없다는 단점이 있다.

주간, 월간 통계의 효율적인 처리

- 일별 통계 테이블(`daily_sales_stats`)만 있으면 주간, 월간, 연간 통계를 별도로 만들 필요가 없다.
- 일별 통계 데이터는 양이 매우 적기 때문에, 이를 다시 `GROUP BY` 하여 월간/연간 통계를 계산해도 비용이 거의 들지 않는다.
- 불필요한 월간/연간 테이블 생성은 관리 포인트를 늘리므로 지양해야 한다.

실시간 통계와 하이브리드 설계

- 실시간 통계가 필요한 경우, **통계 테이블(과거 데이터)**과 **원본 테이블(오늘 데이터)**을 결합하는 하이브리드 방식을 사용한다.
- `UNION ALL`을 사용하여 두 테이블의 데이터를 합쳐서 조회한다.
- 오늘 데이터는 양이 적고 인덱스를 활용할 수 있어 실시간 집계 부하가 적다.
- 대용량 데이터 환경에서도 성능과 실시간성 모두를 확보할 수 있는 전략이다.

멱등성 설계

- 통계 배치는 몇 번을 재실행해도 결과가 같아야 하는 **멱등성(Idempotency)**을 보장해야 한다.
- 기존 값에 더하기를 하는 증분 업데이트(`UPDATE ... + value`) 방식은 재실행 시 데이터 중복(뻥튀기) 문제가 발생하므로 피해야 한다.
- **지우고 다시 쓰기(DELETE & INSERT)** 전략을 사용하면 언제든 배치를 재실행하여 데이터를 복구하거나 과거 데이터 수정(환불 등)을 반영할 수 있다.

마이크로 배치

- 하루 데이터량이 너무 많아 실시간 집계조차 부담스러울 때 사용하는 전략이다.
- 1분, 5분 등의 짧은 주기로 오늘 데이터를 집계하여 별도의 장중 통계 테이블에 저장한다.
- 조회 시에는 '과거 통계 테이블'과 '장중 통계 테이블'을 결합하여 보여준다.
- 실시간성은 약간 떨어지지만(예: 5분 지연), 조회 성능을 극대화하고 DB 부하를 차단할 수 있다.

UPSERT 최적화

- 빈번한 마이크로 배치에서 `DELETE & INSERT`를 사용하면 인덱스 재구축, 로그 생성 등 DB 리소스 낭비가 심하다.
- **UPSERT(`INSERT ... ON DUPLICATE KEY UPDATE`)** 구문을 사용하면 데이터가 없을 땐 넣고, 있을 땐 값만 수정한다.
- UPSERT는 불필요한 삭제와 생성을 방지하여 성능이 뛰어나며, 덮어쓰기 방식이므로 멱등성도 보장된다.
- **일별 배치(새벽)**는 명확한 관리를 위해 `DELETE & INSERT`를, **실시간 마이크로 배치**는 성능을 위해 `UPSERT`를 사용하는 것이 권장된다.

통계 데이터 설계 핵심 정리

- 처음에는 GROUP BY로 시작한다. 데이터가 적을 때는 이것으로 충분하다.
- 성능이 문제되면 일별 통계 테이블을 도입한다. 1년 365행, 10년 3,650행이면 대부분의 조회가 해결된다.
- 주별, 월별 통계는 일별 통계를 GROUP BY하면 된다. 별도 테이블은 대부분 불필요하다.
- 실시간이 필요하면 통계 + 오늘 원본 데이터를 조합한다. 오늘 하루치는 양이 적다.
- 멱등하게 설계한다. 문제가 있으면 쿼리 수정하고 배치 재실행으로 해결할 수 있어야 한다.
- 성능이 중요하면 마이크로 배치를 고민한다.
- UPSERT를 사용하면 SELECT + INSERT/UPDATE 문제를 효과적으로 처리할 수 있다.